

Testing for vulnerabilities (w/o triggering IPS signatures)

Anthony Bettini
McAfee Labs

August 25, 2010



Agenda



- What this presentation is ***NOT***?
- Why is this relevant?
- Vulnerability testing when the patch is already public
 - Window of opportunity
 - Late-stage
- Vulnerability testing for 0days
 - The vulnerability info just went public
 - Self-discovered vulnerabilities
- Conclusions
- Questions and comments



What this presentation is ***NOT***?



- This talk is *not* about:
 - Covert channels
 - Traditional NIPS evasion techniques (like fragmentation)
 - Malware development
 - Product pitches



Why is testing for vulnerabilities relevant?



- To build the best IPS signatures (NIPS and HIPS), we need to understand how attackers can evade them
- The more we know about testing for vulnerabilities, the better job we'll do assessing for vulnerabilities programmatically and responding to vulnerability-centric threats
- Understanding the information gathering techniques can help raise alarms prior to the actual attack stage taking place

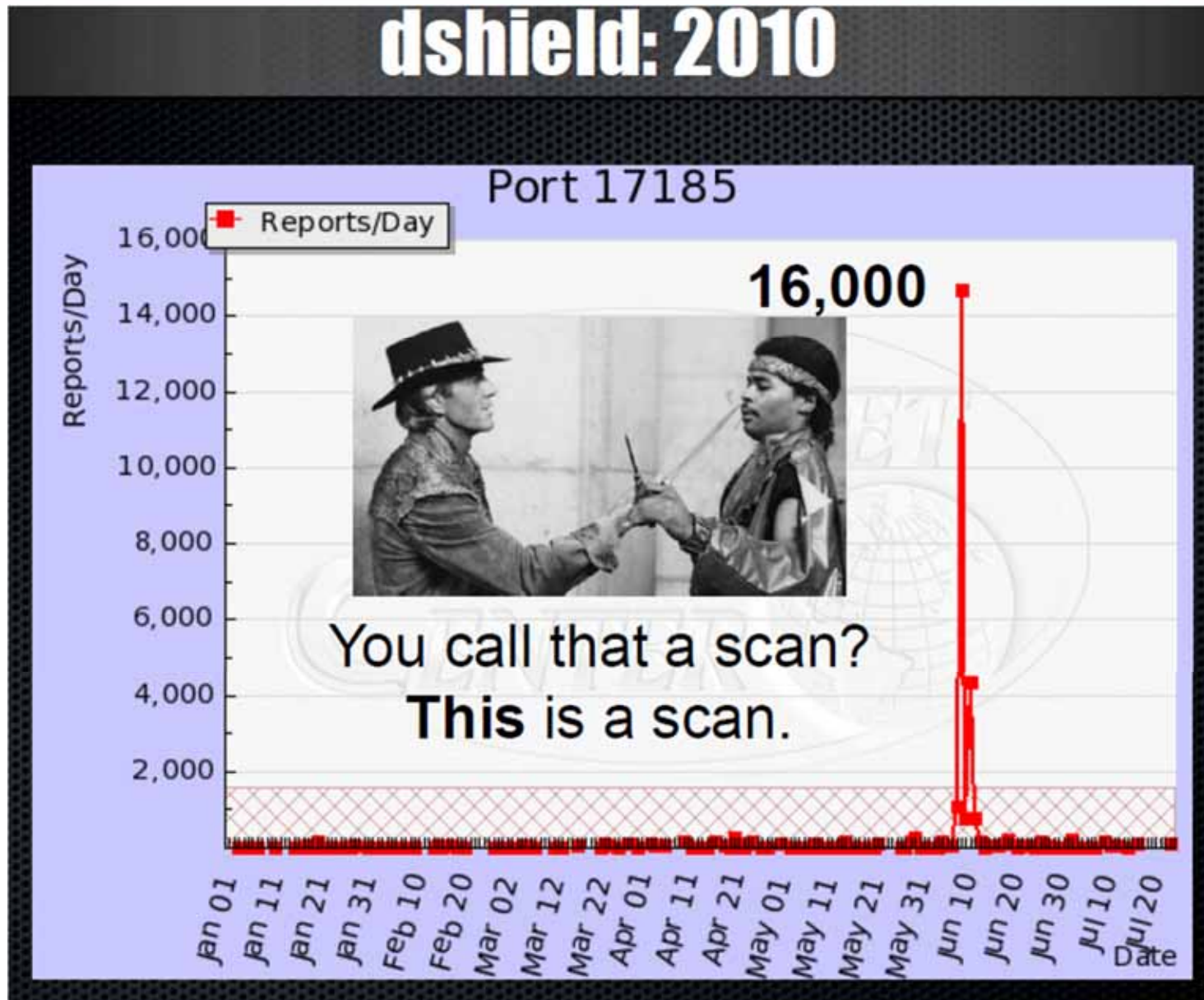


Why do attackers care about this?



- NIPS/HIPS evasion
- Maximizing the effects of attacks while minimizing discoverability
- Leveraging the vulnerabilities pre-patch





Vulnerability testing for *patched* bugs



- Three key types of patched bugs:
 - 1) Critical Microsoft bugs that have patches that are 0-48 hours old (young patches)**
 - 2) Critical Microsoft bugs that have patches that are older than 48 hours
 - 3) All other bugs that have public patches (regardless of severity and vendor)

NOTE: Next few slides uses remote, non-client side bugs, as examples



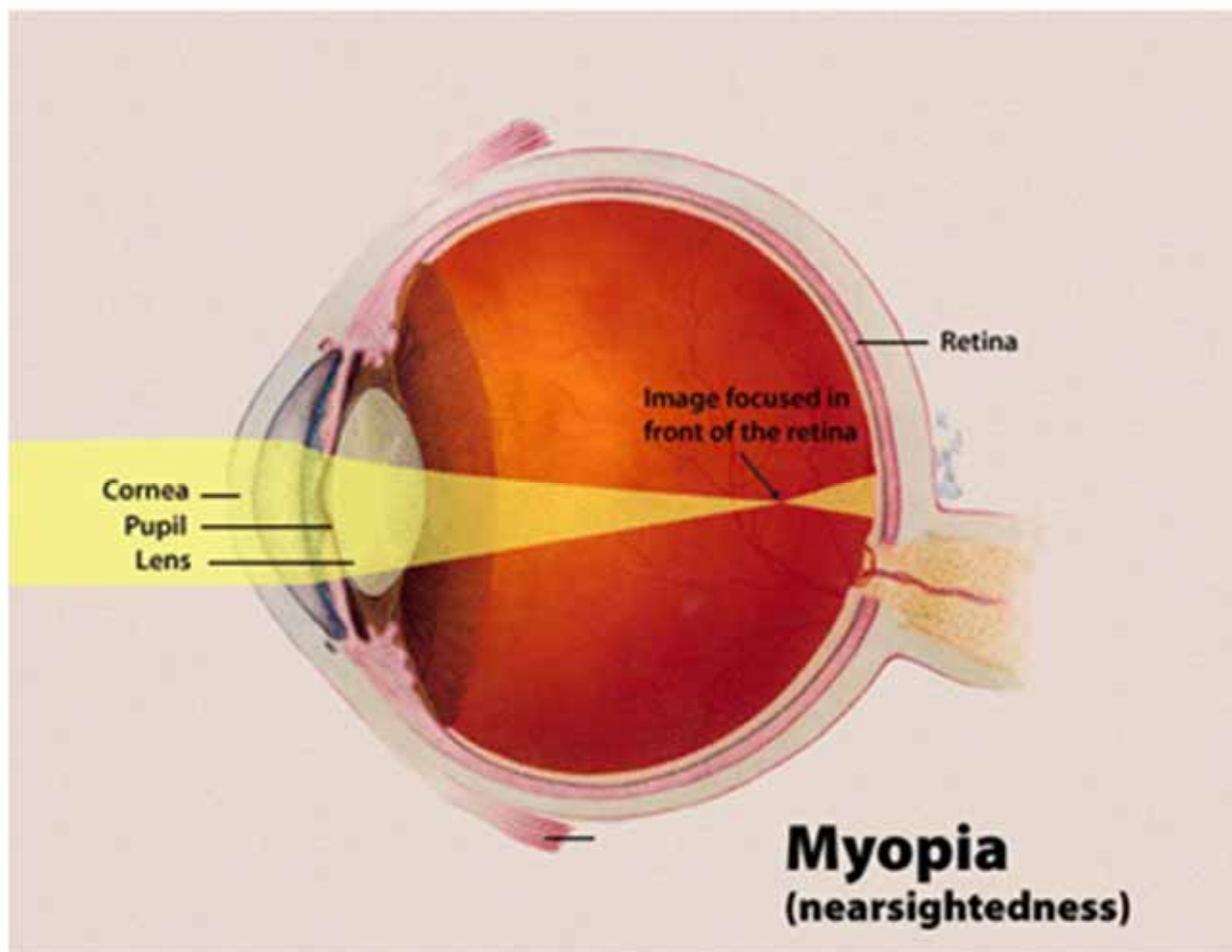
Microsoft?



- Why do we differentiate between Microsoft patches, and all other patches?
 - *Answer: (next slide)*



(In)security myopia



“Those with myopia see near objects clearly but far away objects appear blurred.”



Critical MSFT patches that are 0-48 hours old



- Traits:
 - Unlikely to be patched
 - Mixed results by security mitigations
 - 0 hour mark: no patch deployed, no security mitigation deployed (signatures, updates, etc)
 - 24 hour mark: daily updates deployed, patches mixed on consumer systems and not deployed in the enterprise
 - 48 hour mark: security mitigations likely in-place, mixed patching on both consumer and enterprise systems



Critical MSFT patches that are over 48 hours old



- Traits:

- Likely to be widely patched
- If unpatched, likely will be a botnet participant soon, or already is
- Mixed results by security mitigations
 - 0 hour mark: no patch deployed, no security mitigation deployed (signatures, updates, etc)
 - 24 hour mark: daily updates deployed, patches mixed on consumer systems and not deployed in the enterprise
 - 48 hour mark: security mitigations likely in-place, mixed patching on both consumer and enterprise systems



All other bugs that have public patches



- Traits:

- Patches likely to be deployed very slowly by both consumers and the enterprise
- Enterprise beginning to learn how to patch Adobe vulnerabilities thanks to the Dowd paper and associated fallout



Stereotypical patch deployment summary



- Enterprise:
 - Fairly quick at critical Microsoft patches
 - Scheduled deployments for the rest of the Microsoft patches (long)
 - Very mixed / slow deployment of any other kind of patch
- Consumers:
 - Auto-update
 - If it auto-updates, they have it (includes patches and existing security technologies)
 - May have expired anti-malware solutions that don't auto-update



Vulnerability testing for *unpatched* bugs



- Two key types of unpatched bugs:
 - 1) Publicly reported vulnerability, but for which no public patch exists
 - 2) Self-discovered

NOTE: Due to the laws of secrecy, we include 0days shared amongst private hacker groups as public (item 1)



Publicly reported, but unpatched



- Traits:
 - Will be patched, so the clock is ticking
 - If not-very-critical, perhaps the patch will go public in a year
 - If very critical, perhaps the patch will go public in a week



Self-discovered 0days



- Traits:
 - Could be around a long time
 - How the discoverer handles the 0day, is the strongest determining factor in how long the bug remains private and unpatched



What to look for first?



- Now we know the general behavior of consumers and the enterprise regarding patches
- How does this impact vulnerability testing methodologies for each of the types of bugs we've discussed?



How do we test for the 5 scenarios



- The five we'll cover:
 - 1) Critical Microsoft bugs that have patches that are 0-48 hours old (young patches)
 - 2) Critical Microsoft bugs that have patches that are older than 48 hours
 - 3) All other bugs that have public patches (regardless of severity and vendor)
 - 4) Publicly reported vulnerability, but for which no public patch exists
 - 5) Self-discovered
- Two of the five scenarios very similar (one patched and one unpatched)
 - Critical Microsoft bugs that have patches that are 0-48 hours old (young patches)
 - Publicly reported vulnerability, but for which no public patch exists

Fast-analysis required



```
[*] rpcxenum!1.07!2005!anthony_bettini ... starting analysis
Found __imp__RpcServerRegisterIf@12 0x6fdc994e
Found __imp__RpcServerRegisterIfEx@24 0x6fdd1640

Transfer syntax UUID: 8a885d04-1ceb-11c9-9fe8-08002b104860 v2.0, addr: 0x6fd95400
Interface UUID: 82273fdc-e32a-18c3-3f78-827929dc23ea v0.0, addr: 0x6fd953ec
MIDL Stub Desc addr: 0x6fd95398
num_text_xrefs: 1, num_data_xrefs: 0

Transfer syntax UUID: 8a885d04-1ceb-11c9-9fe8-08002b104860 v2.0, addr: 0x6fd95728
Interface UUID: 50abc2a4-574d-40b3-9d66-ee4fd5fba076 v5.0, addr: 0x6fd95714
MIDL Stub Desc addr: 0x6fd956c0
num_text_xrefs: 1, num_data_xrefs: 0

Transfer syntax UUID: 8a885d04-1ceb-11c9-9fe8-08002b104860 v2.0, addr: 0x6fd96350
Interface UUID: c681d488-d850-11d0-8c52-00c04fd90f7e v1.0, addr: 0x6fd9633c
MIDL Stub Desc addr: 0x6fd962e8
Interface Specification (IfSpec) addr: 0x6feb0f74
num_text_xrefs: 1, num_data_xrefs: 1
DispatchTable addr: 0x6fead850
count: 21
DispatchTable addr: 0x6fd963a0
Opnum 0 (0x0) _EfsRpcOpenFileRaw@16 addr: 0x6fd963a0
Opnum 1 (0x1) _EfsRpcReadFileRaw@8 addr: 0x6fd963a4
Opnum 2 (0x2) _EfsRpcWriteFileRaw@8 addr: 0x6fd963a8
Opnum 3 (0x3) _EfsRpcCloseRaw@4 addr: 0x6fd963ac
Opnum 4 (0x4) _EfsRpcEncryptFileSrv@8 addr: 0x6fd963b0
Opnum 5 (0x5) _EfsRpcDecryptFileSrv@12 addr: 0x6fd963b4
Opnum 6 (0x6) _EfsRpcQueryUsersOnFile@12 addr: 0x6fd963b8
Opnum 7 (0x7) _EfsRpcQueryRecoveryAgents@12 addr: 0x6fd963bc
Opnum 8 (0x8) _EfsRpcRemoveUsersFromFile@12 addr: 0x6fd963c0
Opnum 9 (0x9) _EfsRpcAddUsersToFile@12 addr: 0x6fd963c4
Opnum 10 (0xa) _EfsRpcSetFileEncryptionKey@16 addr: 0x6fd963c8
Opnum 11 (0xb) _EfsRpcNotSupported@28 addr: 0x6fd963cc
Opnum 12 (0xc) _EfsRpcFileKeyInfo@16 addr: 0x6fd963d0
Opnum 13 (0xd) _EfsRpcDuplicateEncryptionInfoFile@28 addr: 0x6fd963d4
Opnum 14 (0xe) _EfsUsePinForEncryptedFiles@12 addr: 0x6fd963d8
Opnum 15 (0xf) _EfsRpcAddUsersToFileEx@20 addr: 0x6fd963dc
Opnum 16 (0x10) _EfsRpcFileKeyInfoEx@24 addr: 0x6fd963e0
Opnum 17 (0x11) _EfsRpcGenerateEfsStream@8 addr: 0x6fd963e4
Opnum 18 (0x12) _EfsRpcGetEncryptedFileMetadata@12 addr: 0x6fd963e8
Opnum 19 (0x13) _EfsRpcSetEncryptedFileMetadata@20 addr: 0x6fd963ec
Opnum 20 (0x14) _EfsRpcFlushEfsCache@4 addr: 0x6fd963f0
```

- Requires fast root cause analysis
 - Can leverage tools like BinDiff for a more perfect analysis, but often cheap manually written tools can be good enough
 - Simple IDC script enumerates an MSRPC interface; MSRPC interfaces are just as likely to change as the procedures
- Using IDC or Python scripts can be a cheap and quick (although less perfect) way to get to the root cause analysis

- Speed is required, as critical MSFT bugs that have patches, and public critical 0days get patched quick
- To leverage the vulnerability, will require speed, to ensure it can be exploited within the 0-48 hour window, prior to wide patch deployment
- Once the root cause analysis is done, then comes the testing/exploitation ...



Another special case ... self-discovered



- Taking advantage of self-discovered bugs, is another special case
- In the aftermath of Code Red, “flash worms”, got a lot of press
- The idea was to:
 - Pre-build a list of vulnerable targets, prior to the vulnerabilities disclosure
 - Programmatically compromise them all using the 0day
 - Be over in a *flash!*
- But while there’s a lot of talk on writing exploits, there isn’t a lot of talk on writing the vulnerability test code (the pre-check), to determine if the host is vulnerable, prior to actually launching the attack
 - Relevant for attackers, due to flash worms (and building more reliable exploits)
 - Relevant for security professionals, to build reliable vulnerability assessment checks that have low false positive and false negative rates
- Self-discovered requires staying as far away from the actual vulnerability as possible; to avoid unnecessary disclosure

Network vulnerability testing techniques



- Common network vulnerability assessment techniques relevant to this discussion:
 - Can be detected in the absence of a patch being publicly available:
 - Port listening
 - Port listening, proper application running, version detection
 - Service listening (may not be port bound), version detection
 - Error code testing along the vulnerable code path
 - Requires the patch to be available to the check writer:
 - Code path changes that are network-accessible and completely unrelated to the vulnerability (but introduced by the patch)



- Advantages:
 - Fast
 - Can be slowed down, and done over a long time
 - Can be worthwhile if the application is bound to a fixed port that is rarely used by anything else (and the application itself is rarely found on any other port)
- Disadvantages:
 - Generally considered inaccurate
 - Useless for applications on varying ports
 - Useless if other applications use that port
 - Doesn't differentiate between vulnerable and not vulnerable, unless all widely deployed versions are vulnerable
 - If not done slowly over time, people will know you're targeting an application on that port (and if you do it this way, it's likely the only application on that port)

Port listening, proper application running, version detection



- Advantages:
 - Usually fast
 - Can be slowed down, and done over a long time
 - Generally accurate, unless the vulnerable module within the application is optional or disabled in some reasonable percentage of cases
- Disadvantages:
 - Version detection may need to be done via application fingerprinting, which will slow down the whole process dramatically (i.e. you can't rely on a banner for an IIS server, but you can send varying HTTP queries to figure out what version of IIS you're interfacing with)
 - Generally considered inaccurate
 - Useless for applications on varying ports
 - Useless if other applications use that port
 - If not done slowly over time, people will know you're targeting an application on that port, and possibly the version, depending on how you write that code

Service listening (may not be port bound), version detection



- Advantages:
 - Can be slowed down, and done over a long time
 - Generally accurate, unless the vulnerable module within the application is optional or disabled in some reasonable percentage of cases
- Disadvantages:
 - Service detection on arbitrary ports will require scanning many more ports and far better application fingerprinting, which will result in a lot of network traffic, noise, and generally slow down the whole process dramatically
 - If not done slowly over time, people will know you're targeting an application on that port, and possibly the version, depending on how you write that code

Error code testing along the vulnerable code path



- Advantages:
 - Can be fast, but not always
 - Very accurate
 - Can be slowed down
- Disadvantages:
 - Generally will disclose the code path to the vulnerability to anyone monitoring the network traffic

Patch-related code path changes



- Advantages:
 - Can be fast, but not always
 - Accurate, traditionally used by network vulnerability scanners
 - Can be slowed down
- Disadvantages:
 - New patches that supersede this patch, can impact future test results
 - Only useful when the patch is already public

- Primary attack scenarios we've covered:
 - Critical Microsoft bugs that have patches that are 0-48 hours old (young patches)
 - Critical Microsoft bugs that have patches that are older than 48 hours
 - All other bugs that have public patches (regardless of severity and vendor)
 - Publicly reported vulnerability, but for which no public patch exists
 - Self-discovered
- Primary vulnerability testing methodologies we've covered:
 - Port listening
 - Port listening, proper application running, version detection
 - Service listening (may not be port bound), version detection
 - Error code testing along the vulnerable code path
 - Code path changes that are network-accessible and completely un-related to the vulnerability (but introduced by the patch)
- Now which ones are vuln testing methodologies match which attacks?

- Self-discovered
 - Port listening, proper application running, version detection
 - Service listening (may not be port bound), version detection
 - Port listening
- The less you disclose in the scanning code the better
- Port listening “good” for things like the recent VxWorks bugs, but not great
- Port/Service listening with version detection is the best methodology in this case, but needs to be coded carefully to disclose as little information about the bug as possible
- Preferred by attackers: Port/Service listening with version detection
- Preferred by vulnerability scanners: N/A

- Critical Microsoft bugs that have patches that are 0-48 hours old (young patches)
- Publicly reported vulnerability, but for which no public patch exists
 - These two both require speed
- Multiple relevant methodologies; bug dependent:
 - Port listening, proper application running, version detection
 - Service listening (may not be port bound), version detection
 - Error code testing along the vulnerable code path
- Preferred by attackers: Port/Service listening with version detection
- Preferred by vulnerability scanners: Error code testing along the vulnerable code path

- Critical Microsoft bugs that have patches that are older than 48 hours
- All other bugs that have public patches (regardless of severity and vendor)
 - These two can both be handled at a more normal/relaxed pace (relatively)
- Multiple relevant methodologies; bug dependent:
 - Port listening, proper application running, version detection
 - Service listening (may not be port bound), version detection
 - Error code testing along the vulnerable code path
 - Code path changes that are network-accessible and completely un-related to the vulnerability (but introduced by the patch)
- Preferred by attackers: Port/Service listening with version detection
- Preferred by vulnerability scanners: Code path changes that are network-accessible and completely un-related to the vulnerability (but introduced by the patch)

- Takeaways and call to action:
 - The threat landscape is and has always been dynamic, don't be caught with your head in the sand
 - Be aware of global threats, don't suffer from organizational myopia
 - The point of entry for an attacker is often the weakest link, it's rarely the front door
 - Attackers could be scanning for 0days now, and we could detect them if we kept our eyes open
 - The VxWorks scan was a good example where existing solutions, like DShield, could have alerted someone, but no one appears to have been looking



Questions and comments



- Testing for vulnerabilities is a challenge, if you need assistance, don't hesitate to ask
- For any additional questions and follow up, I can be reached at:
 - Anthony_Bettini@McAfee.com

